Реализация абстракции в ООП. Абстрактные классы и интерфейсы в Java

1) Уровень абстракции — один из способов сокрытия деталей реализации определенного набора функциональных возможностей. Абстракция данных – выделение значимых характеристик объекта. Позволяет работать с объектом не вдаваясь в особенности реализации.

2) Абстрактное поведение означает что мы знаем, что метод должен что то сделать, но мы не знаем деталей реализации этого метода. Абстрактные методы располагаются в интерфейсах и абстрактных классах.

3) Класс является абстрактным если он содержит хотя бы один абстрактный метод. Наследуюсь от такого класса мы должны переопределить абстрактные методы либо объявить дочерний класс абстрактным.

4) Кроме элементов обыкновенного класса, абстрактный может содержать абстрактные методы.

5) Абстрактный класс может содержать абстрактные методы. Нельзя создать экземпляр такого класса.

6) В java нельзя наследоваться от нескольких классов сразу. Но можно реализовывать много интерфейсов. Интерфейсы же, могут иметь множественное наследование.

7) Программирование на уровне интерфейсов позволяет сделать программу более гибкой, работать с абстракцией а не с реализацией.

8) Интерфейс – контракт обязывающий реализовать все описанное поведение. Можно сказать – уровень абстракции.

9) В интерфейсе не может быть конструкторов, блоков инициализации. Все методы паблик абстракт, кроме дефолтных. Все поля паблик статик final.

10) Final методов в интерфейсе быть не может, это противоречие. Статик методы должны иметь реализацию. Нет смысла писать абстракт, но ошибки не будет.

11) Интерфейсы могут иметь множественное наследование. Если класс реализовывает интерфейс имеющий родителей, то он также должен реализовать методы родителей.

12) Класс, реализующий интерфейсы, обязан реализовать тело каждого метода, либо стать абстрактным.

13) В отличие от абстрактных классов, интерфейсы допускают множественную реализацию. Абстрактные классы позволяют создавать динамические поля и блоки инициализации. Интерфейс может быть функциональным.

14) Невозможно создать объект абстрактного типа.

Перечисляемый тип в Java (since JDK 5.0)

15) enum позволяет создать ограниченный набор static final значений, которые будут singleton. Значения могут иметь поля и методы.

16) enum содержит private конструкторы, это позволяет использовать enum для реализации singleton. Объекты enum имеют ряд методов, такие как ordinal (), name () … Нельзя от него наследоваться и нельзя чтобы enum сам наследовал, так как есть неявное наследование. Switch case.

17) Енамы неявно наследуются от java.lang.Enum. Fields: final String name, final int ordinal. Methods: static valueOf(String)…

18) Можно сказать enum – это класс. enum содержит только приватные конструкторы. От enum нельзя наследоваться.

19) enum может реализовывать интерфейсы.

20) All constructors are private or default. We can use it only within enum itself. Every enum object can override methods, which defined in enum’s body. It looks like anonymous class.

21) Можно статически импортировать объекты enum.

22) enum можно использовать в switch.

23) Instanceof позволяет определить является ли объект экземпляром конкретного класса или экземпляром одного из наследников текущего класса.

24) Ссылочные переменные хранят адрес ячейки памяти, в которой расположен определенный объект. Адрес массива – адрес начального элемента массива. Массив храниться в памяти как неразрывный цельный блок.

SOLID и GRASP принципы

25) Константе можно присвоить значение только один раз. В JAVA есть ключевое слово final.

26) Жесткость возникает если не достаточно уровней абстракции. Fragility means that any changes can brake a program. Tests should figure out fragile areas of code. Needless complexity is areas of code which are not used (YAGNY – you are not gonna need it; KISS – keep it simply stupid). Needless repetition we can circumvent by using DRY(do not repeat yourself).

27) Рефакторинг – это изменение имеющегося кода, добавление новых фич. Для комфортной поддержки кода необходимо следовать принципам SOLID и другим.

28) Инкапсулируйте то что изменяется. Надо выделить компоненты, которые могут измениться, и отделить их от части системы, которая останется неизменной. Инкапсуляция позволит изменить или расширить выделенные компоненты без изменения остальной части системы. Предпочитайте композицию наследованию. При композиции поведение не наследуется, а предоставляется для использования правильно выбранным объектом. Программируйте на уровне интерфейса. Это позволяет сделать код более гибким – паттерн стратегия. Стремитесь к слабой связности взаимодействующих объектов.

Чем меньше объекты знают друг о друге, тем гибче система. Одному компоненту нет необходимости знать о внутреннем устройстве другого. Open\closed. Взаимодействуйте только с близкими друзьями. Это принцип минимальной информированности. При проектировании класса надо обращать внимание на количество классов, с которыми будет происходить его взаимодействие. Чем меньше таких классов, тем гибче система. Класс или метод должен иметь только одну причину для изменения.

29) SOLID – Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle.

30) Single Responsibility Principle значит, что должна быть только одна причина для изменения. На пример класс-контейнер только хранит данные и ни как не обрабатывает. Метод например выполняет только одну операцию но не парсит, не валидирует, не выводит на консоль. Пакет для валидаторов содержит только валидаторы и ничего больше.

31) Open-closed principle значит, что система закрыта от изменений но открыта для дополнений. На пример есть абстрактный класс и мы не будем его изменять, но можем писать сколько угодно реализаций.

32) Liskov Substitution Principle. Наследующий класс должен дополнять, а не замещать поведение базового класса. Экземпляры класса наследника и родителя должны быть взаимозаменяемы.

пример нарушения LSP

```
void drawShape(Shape shape) {
  if (shape instanceof Square) {
     drawSquare((Square) shape);
  } else {
     drawCircle((Circle) shape);
```

```
  }
}
```

тут используется определение класса во время выполнения и в зависимости от результата вызывается нужная функция с явным приведением класса аргумента. И если добавится новый наследник класса Shape, ее нужно будет изменить. В ней используется определение класса во время выполнения и в зависимости от результата вызывается нужная функция с явным приведением класса аргумента. И если добавится новый наследник класса Shape, ее нужно будет изменить. Было бы неплохо прикрутить метод рисования к самим объектам Circle, Square на пример определить абстрактный метод в интерфейсе Sape.

33) Interface Segregation Principle. Клиенты не должны быть вынуждены реализовывать методы, которые они не будут использовать. Вместо большого интерфейса лучше создать много маленьких.

34) Dependency Inversion Principle. Зависимости внутри системы строятся на основе абстракций. Модули программы должны соединяться при помощи абстракций. На пример вместо использования переменной типа класс, можно использовать переменную типа интерфейс.

35) GRASP - общие шаблоны распределения ответственностей. 5 основных и 4 дополнительных шаблона. Information Expert . Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения — информационному эксперту. Creator создает другие объекты. По сути шаблон проектирования Абстрактная фабрика - это альтернатива создателя. Controller призван решить проблему разделения интерфейса и логики в интерактивном приложении. Это не что иное, как хорошо известный контроллер из MVC парадигмы. Контролер отвечает за обработку запросов и решает кому должен делегировать запросы на выполнение. Если обобщить назначение controller, то он должен отвечать за обработку входных системных сообщений. Low coupling говорит о том что необходимо, чтобы код был слабо связан и зависел только от абстракций. Слабая связанность так же встречается в SOLID принципах как The Dependency Inversion Principle (DIP) и слабая связанность по сути это реализация Dependency Injection принципа. High Cohesion очень тесно связанна с Single Responsibility Principle (SRP) с SOLID принципов. High Cohesion получается в результате соблюдения SRP. High Cohesion принцип говорит о том, что класс должен стараться выполнять как можно меньше не специфичных для него задач, и иметь вполне определенную область применения. Pure fabrication. Создание дополнительных объектов таких как сервис или репозиторий. In-direction – паттерн медиатор. Например авиа-диспетчер является посредником между самолётами. Полиморфизм позволяет реализовывать одноименные публичные методы, позволяя различным классам выполнять различные действия при одном и том же вызове. Protected Variations. Шаблон защищает элементы от изменения другими элементами (объектами или подсистемами) с помощью вынесения взаимодействия в фиксированный интерфейс, через который (и только через который) возможно взаимодействие между элементами. Поведение может варьироваться лишь через создание другой реализации интерфейса.

Приёмы объектно-ориентированного проектирования. Шаблоны проектирования

36) Всего есть 23(5 порождающих, 7 структурных и 11 поведенческих) шаблона GOF. Порождающие шаблоны определяют способы создания объектов. Вообще можно сказать, названия говорят сами за себя. Шаблоны проектирования позволяют унифицировать архитектуру. Сделать ее удобной для поддержки в будущем, но на разработку придется потратить больше ресурсов.

37) Стратегия – самый часто используемый шаблон. Объединяет семейство алгоритмов под одним интерфейсом. Состоит из интерфейса и его реализаций. Суть в том что мы используем например переменную интерфейсного типа и можем подставлять в нее разные реализации.

38) Итератор является поведенческим шаблоном. Цель – проход по списку. Состоит из интерфейса содержащего методы: Object next() and Boolean hasNext(). Первый возвращает элемент и переводит указатель, второй проверяет есть ли еще элементы.

39) Decorator is structural pattern. Attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality. Example: we have interface ”Item” which contains a method “void prepare()” . Class “Pizza” implements it. We have a class “PizzaDecorator” which contains a field “Pizza” and implements “Item”. Also there are distinct decorators, which extend “PizzaDecorator” and override the method from interface.

40) Шаблон MVC используется в приложения с графическим интерфейсом. Он определяет разделение ответственности между компонентами программы. User sends a request to a controller. It manipulates with model. It process data. Then model updates view and user sees the result.

41) The Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes. Consists of abstract factory class, factory implementations for various families, interfaces for various products, set of product implementations for various families.

42) Builder is used when the object needs to be created over several steps. It separates the construction of a complex object from its representation. Thereby the same construction process can create different representations. Consists of “ObjectBuilder” interface, concrete implementations of it. Ass well there is a class “ObjectBuilderDirector” and The Object itself.

43) Синглтон гарантирует, что будет создан только один объект. Есть несколько способов реализации этого шаблона. Через перечисление, через обычный класс с полем для хранения объекта и приватного конструктора, а так же нужен будет публичный метод, который проверяет создан ли объект и возвращает имеющийся либо создает новый. А так же можно реализовать с помощью внутреннего класса.

Параметризация (обобщение) типов в Java – Java Generic (since JDK 5.0)

44) Дженерики позволяют выявить многие ошибки на этапе компиляции, предотвращая потенциальные исключения. При компиляции все типы параметров заменяются на Object. И информации о типе, который в него передавали, в классе нет. Внутри параметризированного блока мы можем использовать только методы Object. После компиляции какая-либо информация о дженериках стирается. Это называется "Стирание типов". Стирание типов и дженерики сделаны так, чтобы обеспечить обратную совместимость со старыми версиями JDK.

45) class Exam<T>{}

interface Item<Y>{}

<T> CassName(T t){}

<T> void method(){}

46) int method(<?> param) {…} вопрос заменяет Object и мы можем использовать любой класс, который в любом случае будет происходить от Object.

47) Class Name<T>{ T field;} Вместо T подставится реальный тип, который будет указан при создании объекта класса Name. Объект field будет объектом типа, переданного в параметре типа T. Если в параметре T передать тип String, то экземпляр field будет иметь тип String

48) Можно указать два и более параметров типа через запятую.

```
class Hello<T, V> {
   T ob1;
   V ob2;
   // Передать конструктору ссылки на объекты двух типов
   Hello (T o1, V o2) {
      ob1 = o1;
      ob2 = o2;
```

```
    }
}
```

49) getSum(List<Account> accounts) метод принимает в качестве аргументов только список объектов класса Account. Это ограничение указано в самом методе, в его сигнатуре, программист просто не может передать никакой другой список. Приведение типов (casting) происходит на этапе компиляции. Не нужно приводить явно.

50) <T> с большой буквы

51)

```
//Один обобщенный класс может быть унаследован от другого обобщенного.
//При этом можно использовать различные варианты наследования.
class Account<T>{
    private T Id;
    public Account(T _id){
        Id = _id;
    }
}
//1 - наследник типизирован как базовый
class UniversalAccount<T> extends  Account<T>{
    public UniversalAccount(T id) {
          super(id);  // нужно явно вызвать конструктор базового
    }
}
//2 - создание обычного необобщенного класса-наследника. В этом случае при
//наследовании у базового класса надо явным образом определить используемый тип
class StringAccount extends Account<String>{
    public StringAccount(String id) {
        super(id);
    }
}
//3 - типизация производного класса параметром совсем другого типа
class IntAccount<T> extends Account<Integer>{
    private T Code ;
    public IntAccount(int id){
        super(id);
    }
}
```

52) При переопределении методов в параметризированных классах, в сигнатуре метода нужно указывать параметр дочернего класса.

53) Приведение типов выполняется на этапе компиляции. Когда мы добавляем элемент в параметризированный список происходит неявное приведение.

```
List<String> strings = new ArrayList<String>();
strings.add("abc");
        strings.add( 1); // тут ошибка компиляции


List strings = new ArrayList();
strings.add((String)"abc");
        strings.add((String) 1); //ошибка компиляции
```

54) Когда джава код превращается в байт-код, не сохроняется информация о типах-параметрах. Внутри байт-кода List<Cat> cats не будет отличаться от List<String> strings. В байт-коде ничто не будет говорить о том, что cats — это список объектов Cat. Информация об этом сотрется во время компиляции, и в байт код попадет только информация о том, что у в программе есть некий список List<Object> cats.

55) Методы не могут создавать экземпляры типа-параметра через new. Но есть другие способы.

```
<T> T genericCreate(T a) {
        String s = a.toString();
        T result = (T)s;        // remains String
        // T t = new T();       // нельзя
        Object result2;
        if (a.getClass().getName().contains("Integer"))
          result2 = Integer.valueOf(s);
        else
          result2 = a.getClass().cast(result);
        return (T)result2;
      }
```

Массивы в Java ковариантны

```
String[] strings = new String[] {"a", "b", "c"};
        Object[] arr = strings;
```

«Дженерики» инвариантны.

```
List<Integer> ints = Arrays.asList(1,2,3);
        List<Number> nums = ints; // compile-time error.s;
```

Библиотека контейнеров Java Collections Framework (since JDK 5.0)

56) Коллекции – это структуры для хранения данных разными способами. Коллекции дают больше гибкости и дополнительный функционал по сравнению с массивами. Для разных целей тот или иной способ хранения будет более эффективен.

57) Только ссылочные типы данных.

58) Интерфейс Collection extends Iterable. Класс AbstractCollection implements Collection. И подобная иерархия наблюдается у каждого отдельного интерфейса: List←-----AbstractList; Queue←-----AbstractQueue… В абстрактных классах реализуется общий функционал. А конкретные классы такие как PriorityQueue, ArrayList… реализуют соответствующие интерфейсы и наследуют абстрактные классы.

59) java.util

60) Iterator<E> iterator();

ListIterator<E> listIterator(int index) возвращает объект итератор, у которого есть методы hasNext() and next(). ListIterator позволяет пройти по списку в обоих направлениях.

61) Все коллекции кроме карты реализуют интерфейс Iterable. А также есть Iterator & ListIterator.

62) Iterable знает об Iterator. ListIterator extends Iterator. Iterable содержит метод Iterator iterator(), который возвращает объект Iterator.

63) ConcurrentModificationException возникает при удалении элемента во время обхода в цикле.

64) Iterable allows to use foreach

65)

int size ()

boolean isEmpty ()

boolean contains (Object obj)

boolean containsAll (Collection<?> c)

boolean equals (Object obj)

boolean addAll (Collection<? extends E> from)

boolean remove (Object obj)

boolean removeAll (Collection<?> c)

void clear ()

boolean retainAll (Collection<?> c)

Object[] toArray ()

<T> T[] toArray (T[] arrayToFill)

66) List – обеспечивает доступ по индексу, элементы хранятся в порядке добавления. Set – гарантирует уникальность элементов. SortedSet – гарантирует, что элементы будут отсортированы. NavigableSet – добавляют дополнительный функционал: взятие первого, последнего элементов, поиск минимального и максимального… Queue определят методы для работы с очередью: peek, poll… Deque определяет методы для работы двунаправленной очереди: getLast, getFirst, offerFirst, offerLast…Map определяет методы для работы словаря. SortedMap, NavigableMap обеспечивают упорядоченность элементов и дополнительный функционал для работы в Map.

67) AbstractCollection, AbstractSequentialList, AbstractList, AbstractMap, AbstractSet, AbstractQueue реализуют общие методы для своих интерфейсов. Например не смотря на то что LinkedList and ArrayList реализованы по разному, они имеют совершенно одинаковые методы indexOf(Object o), clear().

68) ArrayList основан на массиве. Взятие элемента Log(1). Вставка, удаление, добавление log(n). Используем когда известно количество элементов и список будет редко изменяться. DEFAULT_CAPACITY = 10. newCapacity = oldCapacity + (oldCapacity >> 1);

LinkedList основан на двусвязном списке. Содержит поля с первым и последним элементом. Вставка, удаление log(1). Взятие элемента и поиск log(n). Но если вставлять в конец или начало, то будет быстрее.

69) Коллекции по сравнению с массивом позволяют работать с абстракцией List. Размерность массива нельзя изменить. ArrayList используем когда известно количество элементов и список будет редко изменяться. LinkedList используем когда часто добавляем, удаляем.

70) Цикл for, foreach, метод foreach(), взятие итератора. ListIterator позволяет проходить в обе стороны.

71) Iteator только в одну сторону, ListIteratior в обе стороны.

72) Реализация Set гарантирует уникальность элементов. HashSet реализован на основе HashMap, Когда мы добавляем новый элемент в HashSet он добавляется вместо ключа в HashMap, а вместо значения объект-заглушка. В свою очередь HashMap Реализован на основе асоциативного массива. В каждой ячейке массива содержится ссылка на связанный список, это необходимо если хеш-коды ключей совпадут. LinkedHashSet сохраняет порядок добавления элементов. TreeSet основан на красно-черном дереве и поддерживает упорядоченность, для сравнения объектов используется Comparator and Comparable.

73) В HashSet берется хешкод и по этому хешкоду элемент размещается в ассоциативном массиве. Если хешкоды совпадают тогда проверка на equals если различны, то по одному хешкоду будет несколько элементов в связном списке. Если равны то замена (по сути ничего не изменится для пользователя). TreeSet поддерживает упорядоченность, для сравнения объектов используется Comparator and Comparable. Переопределенные методы возвращают положительное, отрицательное число либо ноль. <, >, =.

74) We can not add null into TreeSet as we can not call method on null pointer(compareTo()).

75) Очереди работают по принципу – первый вошел, первый вышел. LinkedList implements Queue. А также PriorityQueue implements Queue. Очередь с приоритетом значит, что элементы будут сравниваться

при добавлении и размещаться по очереди но среди элементов равных с ними. Сравнение осуществляется по comparator & Comparable. Deque – двунаправленная очередь, элементы можно брать и добавлять в обе стороны. Есть две реализации – на основе LinkedList and ArrayDequeue, названия описывают реализацию. LinkedList потребляет больше памяти чем ArrayDequeue а также больше ресурсов для итерации. Единственная лучшая операция связанного списка - удаление текущего элемента во время итерации. Vector – устаревшая структура, основанная на массиве, все методы синхронизированы. Stack работает по принципу первый вошел – последний вышел, методы peek & poll.

76) HashMap- при добавлении элемента берется хеш-код ключа и элемент помещается в ассоциативный массив в ячейку соответсвующую хеш-коду. Если хеш-коды совпадают тогда ключи проверяются по equals, если все-таки ключи различны, то новый элемент помещается в связный список, таким образом одному хеш-коду соответствует список содержащий несколько элементов. Поиск, вставка, удаление – среднее(log(1)), худшее(log(n)). LoadFactor = 0,75. DefaultCapasity = 16. LinkedHashMap поддерживает порядок добавления. TreeMap поддерживает по Comparator & Comparable.

77) Произойдет замена старого значения на новое.

78) У потокобезопасных контейнеров синхронизированные методы. Vector позволяет брать элементы по индексу, элементы хранятся в порядке добавления. Stack предоставляет возможность хранения по модели стек – первый вошел, последний вышел. HashTable – аналог HeshMap, но не может хранить null.

79) ArrayList не потокобезопасен в отличие от Vector и Stack, соответственно работает быстрее.

80) HashTable (synchronize) – аналог HeshMap, но HashTable не может хранить null. Не рекомендуется использовать HashTable даже в многопоточных приложениях.

81) Collections – collection service. Предоставляет дополнительный функционал для работы с коллекциями. Arrays – соответсвенно. binarySearch(byte[] a, byte key), copyOf(boolean[] original, int newLength), deepEquals(Object[] a1, Object[] a2), sort(int[] a), toString(int[] a)…

82) Что бы отсортировать коллекцию, необходимо чтобы ее элементы реализовывали Comparator or Comparable. ArrayList, LinkedList, LinkedHashSet… - элементы хранятся в порядке добавления. Tree-HashMap, TreeSet… - элементы хранятся в отсортированном виде. HashMap, HashSet…- в случайном порядке.

83) Comparable.

84) Если у нас есть доступ к коду класса, то можно реализовать Comparable. Если доступа нет, то нужно создать дополнительный класс, который реализует Comparator и передать объект этого класса в конструктор коллекции.

85) Создать объект UnmodifiableSortedSet<>(set); UnmodifiableList<>(list)…

86) Стратегия – самый часто используемый шаблон. Объединяет семейство алгоритмов под одним интерфейсом. Состоит из интерфейса и его реализаций. Суть в том что мы используем например переменную интерфейсного типа и можем подставлять в нее разные реализации.

Основы потоков ввода-вывода в Java

87) Файл- это именованная область данных на носителе информации. Атрибут файла – это характеристика, описывающая файл, например скрытый или только для чтения.

88) Файловая система — порядок, определяющий способ организации, хранения и именования данных на носителях. Файловая система позволяет создавать, читать, искать, защищать, удалять файлы.

89) Бинарные и текстовые.

90) Здесь поток – это абстракция, которая используется для чтения или записи информации в файл, консоль или сокет. Есть потоки ввода и вывода.

91) Физически, поток –это последовательность данных.

92) Java.io; java.nio.

93) Decorator pattern.

94) Ввод-вывод; Текстовые и бинарные. Буферизированные и нет.

95) На самом низком уровне иерархии потоков ввода-вывода находятся потоки, оперирующие байтами. Это объясняется тем, что многие устройства при выполнении операций ввода-вывода ориентированы на байты.

96) Closeable, Flushable – функциональные интерфейсы, содержащие мето-ды close & flush.close() – закрывает поток. Flash() – выводит все из буфера.

97) Древо IOExeption является checked так как происходит взаимодействие с операционной системой, неконтролируемой jvm. FileNotFoundException, EOFException…

98) Предоставляют функционал для чтения и записи данных в байтовом виде. Класс InputStream: int read() – читает один байт либо возвращает -1. Int read(byte[] buffer) считывает байты по размеру массива…

99) InputStream implements Closeable, OutputStream implements Closeable, Flushable – абстрактные классы. Определяют методы read & write с различными перегрузками.

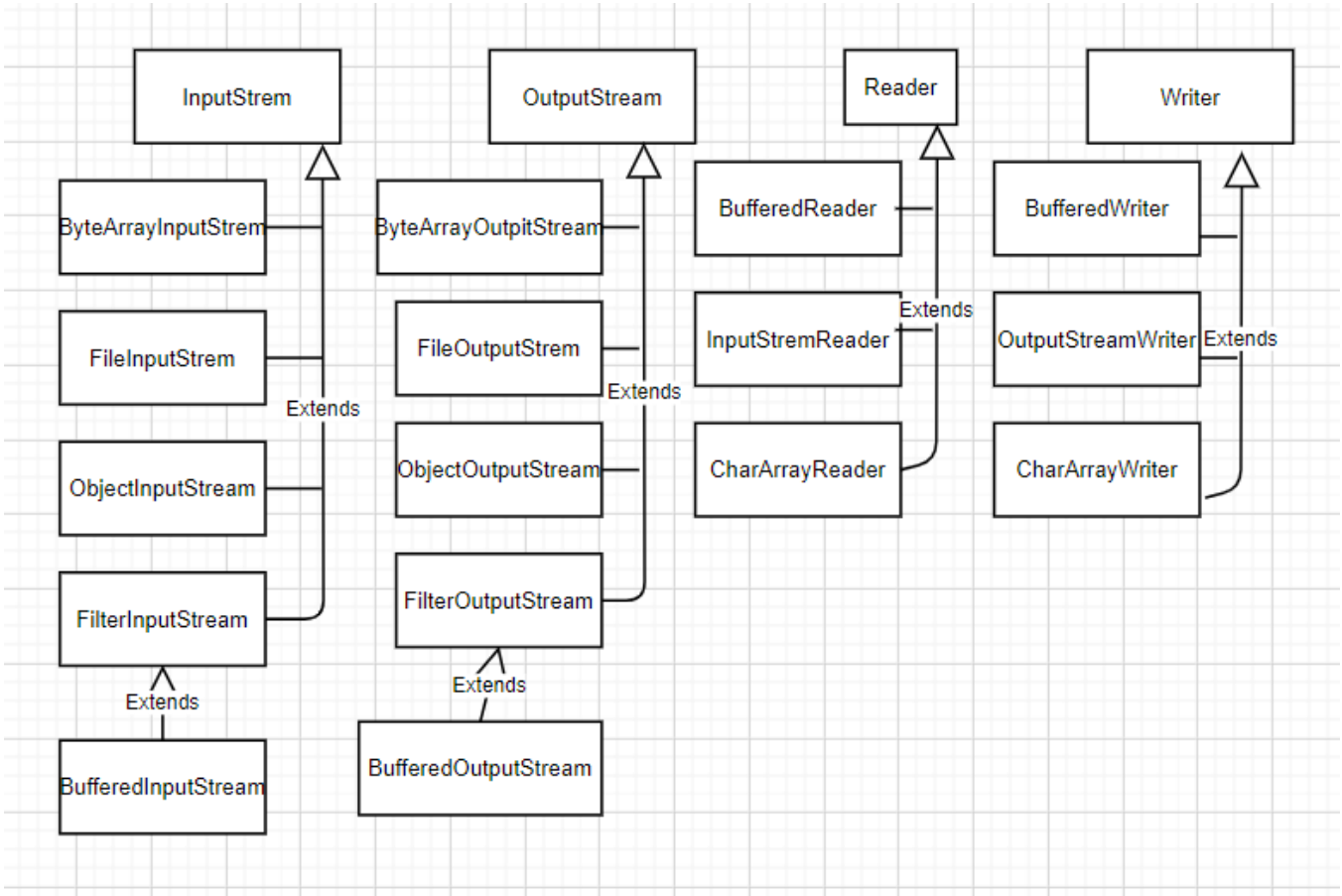
100) Предоставляют функционал для чтения и записи данных посимвольно.

101) Unicode

102) abstract class Reader implements Readable, Closeable ; abstract class Writer implements Appendable, Closeable, Flushable; Определяют методы read & write с различными перегрузками.

103) Байтовые работают с байтами (int), поэтому могут понадобится дополнительные преобразования. Символьные могут работать с char. И те и те имеют одинаковые имена методов и реализуют одни и те же интерфейсы.

104)



105) Шаблоны Mediator, Interpreter, Proxy, Adapter. Если имелось ввиду переложить шаблоны проектирования на взаимодействие разработчиков.

106) java.util.Scanner считывает данные из строки, из файла, из консоли (объект чтения передается в конструктор). nextLine() читает строку. nextInt() считывает и возвращает введенное число, может вызвать java.util.InputMismatchException. hasNextInt() — метод проверяет, является ли следующая порция введенных данных числом

107) Сериализация – это запись объекта (только полей) в поток, что позволяет вывести жизненный цикл объекта за рамки жизненного цикла программы.

108) Десериализация – это восстановление объекта из потока.

109) В процессе сериализации вместе с сериализуемым объектом сохраняется его граф объектов. В java есть несколько способов сериализации. Реализовать интерфейс Serializable, который не содержит методов. Создать объект ObjectOutputStream например так: ObjectOutputStream objectOut-putStream = new ObjectOutputStream(new FileOutputStream("fileName.out")); далее вызвать метод writeObject и передать в него записываемый объект. Ключевое слово transient применя-ется к полям и исключает поле из сериализации. При десериализации эти поля будут заполнены по умолчанию. А так же что бы сделать объект сериализуемым можно реализовать интерфейс Externalizable – методы writeExternal(ObjectOutput out) и readExternal(ObjectInput in). В этом случае обязателен пустой конструктор.

Основы параллельного (многопоточного) программирования в Java. Потоки выполнения

110) Многопоточность бывает реальная (многоядерный процессор) и виртуальная (одноядерный процессор, процессорное время дается потокам по решению планировщика по очереди). Многопоточность основанная на поток и многопоточность основанная на процессах. А так же одна программа (состоит из нескольки потоков) может обрабатываться несколькими ядрами процессора, что повышает скорость. Процесс – выполняемая программа, элемент кода, которым может управлять планировщик. Поток – наименьший элемент управляемого кода. Потоки бывают обыкновенные и демоны.

111) Процесс работы jvm состоит из нескольких потоков, часть из них обеспечивают работу самой jvm, а часть создает программист. Планировщик выделяет квант времени на процесс, и далее этот квант времени распределяется внутри процесса на потоки. Потоки имеют приоритеты от 1 до 10. По умолчанию приоритет 5.

112) Поток – наименьший элемент управляемого кода. Потоки бывают обыкновенные и демоны. Демон – это фоновый поток, программа не зависит от него на прямую, например запись в лог. А так же есть главный поток и вызванные им дочерние потоки. Потоки имеют приоритеты от 1 до 10. По умолчанию приоритет 5. У потоков есть имена и состояния (New, Runnable, Running, Waiting/blocked/ sleeping, Dead).

113) Есть главный поток и вызванные им дочерние потоки. Thread.currentThread().

114) Объект потока создан(New), вызван метод start(Rannable), планировщик дал процессорное время(Running), планировщик отдал процессорное время другому, текущий переходит в (Waiting), вызван метод sleep, поток переходит в sleeping, поток закончил спать(Running), поток завершился – dead.

115) Унаследовать Thread и переопределить метод run. Либо реализовать Runnable, создать объект Thread и в конструктор передать объект реализовавший Runnable (можно на лету). Отличие этих двух методов заключается в том, что реализация Runnable является более гибкой и позволяет кроме реализации интерфейса еще и унаследоваться от какого-нибудь другого класса.

116) Id – номер потока. Название – имя потока, у главного потока имя main, имена могут назначаться автоматически. Потоки имеют приоритеты от 1 до 10. По умолчанию приоритет 5. А так же определены константы с приоритетами. Группу можно передать в конструктор. States: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED.

117) Планировщик раздает процессорное время непредсказуемо. Приоритет потока не пропорционален реально выдаваемому времени, есть некий примерный коэффициент. Реальная разница между малым и высоким приоритетом не отличается в 10 раз! Она не такая значительная.

118) Поток может оказывать влияние только на потоки, которые находятся в одной с ним группе. Группу потоков представляет класс ThreadGroup. Такая организация позволяет защитить потоки от нежелательного внешнего воздействия. Группа потоков может содержать другие группы, что позволяет организовать все потоки и группы в иерархическое дерево, в котором каждый объект ThreadGroup, за исключением корневого, имеет родителя. Класс SecurityManager указывает ограничения доступа к определенным группам потоков.

119) Потоки имеют приоритеты от 1 до 10. По умолчанию приоритет 5. А так же есть константы с приоритетами. Приоритет потока не пропорционален реально выдаваемому времени, есть некий примерный коэффициент. Реальная разница между малым и высоким приоритетом не отличается в 10 раз! Она не такая значительная.

120) Пользовательские потоки являются приоритетными. JVM будет ждать, пока последний fore-ground thread завершит свою задачу, прежде чем завершить его. Если все главные потоки завершились, но остались демоны, jvm завершит работу. Демон – это фоновый поток, программа не зависит от него на прямую, например запись в лог. Чтобы сделать поток демоном, нужно до запуска потока вызвать setDaemon(true); isDeamon() – проверяет является ли поток демоном.

Основы синхронизации потоков в Java. Использование библиотеки java.util.concur-rency

121) Состояние гонки возникает, когда один и тот же ресурс используется несколькими потоками одновременно, и в зависимости от порядка действий каждого потока может быть несколько возможных результатов. Гонка данных возникает, когда два или более потока пытаются получить доступ к одной и той же не финальной переменной без синхронизации.

122) Для синхронизации используется ключевое слово synchronized. Оно может быть установлено в сигнатуре метода либо в критическом блоке с указанием объекта на котором синхронизируемся. Этот объект должен быть общий для всех потоков. Когда один поток зашел в синхронизированный блок, другие потоки не смогут зайти в этот же блок пока текущий поток не завершит этот блок. Другие потоки будут ждать или выполнять другой код пока не освободится заблокированный.

123) Объект потока создан(New), вызван метод start(Rannable), планировщик дал процессорное время(Running), планировщик отдал процессорное время другому, текущий переходит в (Waiting), вызван метод sleep, поток переходит в sleeping, поток закончил спать(Running), поток обратился к синхронизированному блоку, блок оказался занят – поток ждет. Блок освободился, поток успел занять этот блок и объявил его заблокированным. Поток выполняет код в синхронизированном блоке и тут процессорное время отдается другому потоку – другой поток может выполнить какой-нибудь другой код, но в текущий синхронизированный блок все равно зайти не может. Потом первому потоку снова дается процессорное время, он завершает синхронизированный блок и объявляет его свободным. Поток завершился – dead.

124) Монитор – это флаг, показывающий свободен ли синхронизированный блок или заблокирован другим потоком. А так же должен всегда использоваться объект синхронизации. При объявлении метода synchronized объектом блокировки будет объект класса. При создании синхронизированного блока можно использовать статический объект, можно передавать объект синхронизации в метод, можно this…

125) Реализация синхронизации осуществляется с помощью key word synchronized и объекта синхронизации. Если объект в синхронизированном блоке null тогда броситься NullPointerException.

126) Для синхронизации используется ключевое слово synchronized. Оно может быть установлено в сигнатуре метода либо в критическом блоке с указанием объекта на котором синхронизируемся. Этот объект должен быть общий для всех потоков. При объявлении метода synchronized объектом блокировки будет объект класса. При создании синхронизированного блока можно использовать статический объект, можно передавать объект синхронизации в метод, можно this…

127) Deadlock очень тяжело предупредить и такое может происходить даже в качественном коде. На пример есть два потока и они оба вошли в свои синхронизированные блоки, потом один поток обращается к синхронизированному методу другого потока, а тот в свое время вызывает синхронизированный метод первого. Таким образом они будут ждать друг друга. Если это произошло, то придется удалять потоки пока программа не продолжит работать, ведь хуже не станет.

128) Атомарная операция –это простейшая операция, у которой нет промежуточных, то есть либо значение до либо после. В java операции записи и чтения всех полей кроме long & double являются атомарными. Это по тому что некоторые 32-битные платформы не могут атомарно читать 64-битное значение. Могут возникнуть проблемы если один поток читает а второй записывает. Потоки кешируют

поля классов и получается, что каждый поток работает со своим значением. Эти две проблемы решает слово volatile.

129) Блокирующая синхронизация заставляет другие потоки ждать и ничего не выполнять. Неблокирующая позволяет выполнять другие операции пока ждем освобождения критического блока.

130) ReentrantLock реализует интерфейс Lock и имеет дополнительные возможности, связанные с опросом о блокировании.

```
static Lock lock = new ReentrantLock();
@Override
public void run() {
    while (true) {
        if (lock.tryLock()) {
            printer.print(text);
            lock.unlock();
            break;
        }
        try {
            TimeUnit.MILLISECONDS.sleep(1800);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("\n" + text + " is doing something other");
    }
}
```

Нужно создать объект ReentrantLock в поле класса и на нем вызвать метод tryLock() если блок не занят то запустить выполнение, а потом unlock(); Если блок занят пойти дальше.